# Linearly Refined Session Types

Pedro Baltazar Dimitris Mostrous Vasco T. Vasconcelos
University of Lisbon, Faculty of Sciences and LaSIGE
Lisbon, Portugal
{pbtz,dimitris,vv}@di.fc.ul.pt

Session types capture precise protocol structure in concurrent programming, but do not specify properties of the exchanged values beyond their basic type. Refinement types are a form of dependent types that can address this limitation, combining types with logical formulae that may refer to program values and can constrain types using arbitrary predicates. We present a pi calculus with assume and assert operations, typed using a session discipline that incorporates refinement formulae written in a fragment of Multiplicative Linear Logic. Our original combination of session and refinement types, together with the well established benefits of linearity, allows very fine-grained specifications of communication protocols in which refinement formulae are treated as logical resources rather than persistent truths.

## 1 Introduction

*Session types* [9] are a practical and expressive type-based verification methodology for concurrent programming, and have proved excellent in modeling typed computations predominantly consisting of client-server message passing. As a simple example, assigning the type $!\mathsf{int}.?\mathsf{bool}.\mathsf{end}$ to a communication channel means that a value of type $\mathsf{int}$ will be sent ($!\mathsf{int}$), then a $\mathsf{bool}$ will be received ($?\mathsf{bool}$), and the channel cannot be used any further. Communication soundness is ensured when the "other end" of the communication channel is used in a complementary (or dual) way: $\overline{!\mathsf{int}.?\mathsf{bool}.\mathsf{end}} = ?\mathsf{int}.!\mathsf{bool}.\mathsf{end}$.

*Refinement types*, as defined for ML [6], are a form of dependent types that allow the programmer to attach formulae to types, thus narrowing down the set of values inhabiting a given type. For instance, the type $\{x : \mathsf{int} \mid 0 \leq x \wedge x \leq 10\}$ describes integer values in the range 0..10. Such fine-grained types have met increasing attention, with several notable works on type checking for functional programming, such as hybrid type checking [5], liquid types [16], or the blame calculus [19]. In the context of this work, let us note that refinements for ML written in Intuitionistic Linear Logic have been introduced in [11]. A detailed overview is in [8].

With regard to refinement formulae, the most common approach is to use classical first-order logic, which is certainly enough for many examples, but cannot provide a satisfactory treatment of refinements on resources. In particular, it does not allow one to control finer computational properties: a type $\{x\colon \mathsf{ccard} \mid \mathsf{use}(x)\}$ may mean that we can use a credit card, but it does not mandate that we can do so *just once*. To achieve such finer distinctions between types, we specify refinements in a fragment of *multiplicative linear logic* (MLL) [7], most notably without exponentials or additives.

Building on previous work on session types [18], we combine sessions and linear refinements to obtain an original system of *linearly refined session types*, noting that until now neither linear nor classical refinements have been studied in the context of session types, according to our knowledge. The result is a system in which typed message exchange, refinement, and resources are combined, providing for a very fine control of *process behaviour*. We show that well-typed programs do not get stuck when trying to verify logical properties.

S.Alves and I. Mackie (Eds.): Second International Workshop
on Linearity 2012 (LINEARITY'12).
EPTCS 101, 2012, pp. 38–49, doi:10.4204/EPTCS.101.4

The rest of the paper is structured as follows. Section 2 introduces the language and its operational semantics, as well as the running example. Section 3 describes the typing system and Section 4 the main results. We conclude the paper with related work and future directions.

## 2 The pi calculus with assume and assert

Consider a simple online Store that accepts a product request from a Client, and interacts with a Bank to perform the payment. The Store and the Client share a private channel in which the Client sends the product p, the credit card number c and the price it is willing to pay, €100. The Store acts dually by accepting the product, the credit card, and the amount to be charged, and by immediately charging, using $\mathbf{Charge}(c,a)$, the amount $a$ to the credit card $c$.

$$\mathbf{Client} = s_1!\mathrm{p}.s_1!\mathrm{c}.s_1!100.\mathbf{0} \qquad \mathbf{Store} = s_2?p.s_2?c.s_2?a.\mathbf{Charge}(c,a)$$

In the code above, $s_1!$p means to send the value p on channel endpoint $s_1$, dually $s_2?p$ means to read a value from $s_2$ binding it to variable $p$, and $\mathbf{0}$ denotes the terminated process. Value p should not be confused with variable $p$. In our language processes read and write within sessions by using distinct variables to identify the two ends of the channel, $s_1$ and $s_2$ in this case.

In order to charge the Client, the Store calls the Bank service, and sends the credit card number and the amount to be charged.

$$\mathbf{Bank} = *r_1?y.y?c.y?a.\mathbf{0} \qquad \mathbf{Charge}(c,a) = (\mathsf{new}\ b_1b_2)(r_2!b_2.b_1!c.b_1!a.\mathbf{0})$$

In the Bank code, $*$ prefixes a replicated process that can be used an unbounded number of times, as one would expect in this example. The **Charge** process creates a new channel with the $(\mathsf{new}\ b_1b_2)$ constructor, whose purpose is to establish a private, bidirectional channel with the bank. To set up the session, the channel endpoint $b_2$ is passed to the bank and the other, $b_1$, is retained locally for interaction with the bank. Note that the language is explicitly typed, but for brevity we ignore the type annotations in our examples.

The overall system is the parallel composition of the three processes connected by two channels: $r_1r_2$, the public Bank-Store channel, and $s_1s_2$, the private Client-Store channel.

$$(\mathsf{new}\ r_1r_2)(\mathsf{new}\ s_1s_2)(\mathbf{Client} \mid \mathbf{Store} \mid \mathbf{Bank})$$

The syntax of processes is presented in Figure 1. The linear nature of sessions, for example in the session $s_1s_2$ between Client and Store, can ensure some security properties. By enriching such a calculus with cryptography primitives, more properties can be captured, such as authentication requirements and privacy of communication (e.g. [2]). However, even if such properties are satisfied the system can contain unintended uses of given permissions by authorized processes. In the above example, the Store can wrongly compute the amount to be charged, which will be detected only later by the Client.

$$\mathbf{Store_1} = s_2?p.s_2?c.s_2?a.\mathbf{Charge}(c,a+10)$$

A more subtle situation is when two threads try to charge the Client for the same purchase.

$$\mathbf{Store_2} = s_2?p.s_2?c.s_2?a.(\mathbf{Charge}(c,a) \mid \mathbf{Charge}(c,a))$$

Our language enriches pi calculus with assume and assert commands, using formulae $\varphi$ built over a set of uninterpreted predicates $A, A_1, A_2, \ldots$, the linear logic connective of tensor, $\otimes$, and its identity, $\mathbf{1}$.

| | | | | | |
|---|---|---|---|---|---|
| $\varphi$ ::= | | *Formulae:* | $P$ ::= | | *Processes:* |
| | $A(v_1,\ldots,v_n)$ | predicate on $v_1,\ldots,v_n$ | | $x!v.P$ | output |
| | $\varphi \otimes \varphi$ | joining | | $x?x.P$ | input |
| | $\mathbf{1}$ | identity | | $P \mid P$ | parallel composition |
| | | | | $*P$ | replication |
| $v$ ::= | | *Values:* | | $\mathbf{0}$ | inaction |
| | $x$ | variable | | $(\mathsf{new}\ xx\colon T)P$ | scope restriction |
| | $()$ | unit | | $(\mathsf{assume}\ \varphi)P$ | assume |
| | | | | $\mathsf{assert}\ \varphi.P$ | assert |

Figure 1: The syntax of processes

The predicates may refer to channel names or base-values such as integers and strings, which are represented here by the unit value, $()$; therefore, refinements form dependent types. Enhanced with these commands, the Client may assume a $\mathit{charge}(\mathrm{c},100)$ capability on the values sent to the Store. And the Bank, in turn, will assert that exact capability.

$$\mathbf{Client_1} = (\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))\mathbf{Client} \qquad \mathbf{Bank_1} = *r_1?y.y?c.y?a.\mathsf{assert}\ \mathit{charge}(c,a).\mathbf{0}$$

In order to explain the interplay between assume and assert, we turn our attention to the operational semantics of the language. We say that variable $y$ occurs *bound* in process $P$ within $x?y.P$ and $(\mathsf{new}\ xy\colon T)P$, in type $U$ within $q?y\colon T.U$ and $q!y\colon T.U$, and in formula $\varphi$ within $\{y\colon T|\varphi\}$. Also, variable $x$ occurs bound in $(\mathsf{new}\ xy\colon T)P$. A variable that occurs in a non-bound position within a process, type, or formula is said to be *free*. The sets of free variables in a process $P$, a type $T$ or a formula $\varphi$, denoted by $\mathrm{fv}(P)$, $\mathrm{fv}(T)$ and $\mathrm{fv}(\varphi)$, are defined accordingly and so is alpha-conversion. We work up to alpha-conversion and follow Barendregt's variable convention, whereby all variables in binding occurrences in any mathematical context are pairwise distinct and distinct from the free variables.

The standard capture-free *substitution* of variable $x$ by value $v$ in process $P$, a type $T$ or a formula $\varphi$, is denoted by $P[v/x]$, $\varphi[v/x]$ and $T[v/x]$. This follows the standard treatment for dependent session/channel types [12, 20]. For example, the substitution $((\mathsf{new}\ xy\colon T)P)[v/z]$ is defined as $(\mathsf{new}\ xy\colon T[v/z])P[v/z]$.

From the operational semantics we factor out a *heating relation* meant to simplify the statement of the reduction relation, by structurally adjusting processes. Both relations, heating and reduction, are defined in Figure 2. We start with reduction. The relation includes the rule for communication, R-COM, adapted from [18] to handle dependent refinements, and the usual rules for reduction underneath parallel composition and restriction, R-PAR, R-RES, and under heating with R-HEAT. It also includes two novelties: an axiom R-ASSERT for cutting assertions, and a rule that allows reduction under assumptions, R-ASSUME. The correspondence of R-ASSERT with the logical cut is evident, noting that a choice has been made for assumptions to define a scope and for the cut to take place against enclosed assertions. The alternative would be for the cut to take place between an assume and an assert in parallel, but at the typing level this would require a form of negation which would effectively identify assumptions and assertions; instead of asserting $\varphi$ one could assume $\varphi^{\perp}$ and the two possibilities would be indistinguishable at the typing environment level. As a result, two assumes could cancel out each other, and similarly for two asserts, thus compromising the intended usage of assertions.

*Heating relation,* $P \Rrightarrow Q$ ($P \equiv Q$ means $P \Rrightarrow Q$ and $Q \Rrightarrow P$)

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P \qquad *P \equiv P \mid *P \qquad (\mathsf{new}\ xy\colon T)\mathbf{0} \equiv \mathbf{0}$$
$$(\mathsf{new}\ xy\colon T)(P \mid Q) \equiv (\mathsf{new}\ xy\colon T)P \mid Q \qquad (\mathsf{new}\ wz\colon T)(\mathsf{new}\ xy\colon U)P \equiv (\mathsf{new}\ xy\colon U)(\mathsf{new}\ wz\colon T)P$$
$$(\mathsf{new}\ xy\colon T)(\mathsf{assume}\ \varphi)P \equiv (\mathsf{assume}\ \varphi)(\mathsf{new}\ xy\colon T)P \qquad (\mathsf{assume}\ \mathbf{1})P \equiv P \qquad \mathsf{assert}\ \mathbf{1}.P \equiv P$$
$$(\mathsf{assume}\ \varphi_1)(\mathsf{assume}\ \varphi_2)P \equiv (\mathsf{assume}\ \varphi_2)(\mathsf{assume}\ \varphi_1)P \qquad \mathsf{assert}\ \varphi_1.\mathsf{assert}\ \varphi_2.P \equiv \mathsf{assert}\ \varphi_2.\mathsf{assert}\ \varphi_1.P$$
$$\mathsf{assert}\ \varphi_1 \otimes \varphi_2.P \equiv \mathsf{assert}\ \varphi_1.\mathsf{assert}\ \varphi_2.P \qquad (\mathsf{assume}\ \varphi_1 \otimes \varphi_2)P \equiv (\mathsf{assume}\ \varphi_1)(\mathsf{assume}\ \varphi_2)P$$
$$(\mathsf{assume}\ \varphi)P \mid Q \Rrightarrow (\mathsf{assume}\ \varphi)(P \mid Q) \qquad (\mathsf{new}\ xy\colon T)P \equiv (\mathsf{new}\ xy\colon U)P \quad \text{if } T \equiv U$$

*Reduction relation,* $P \to Q$

$$(\mathsf{new}\ xy\colon (q!w\colon T.U))(x!v.P \mid y?z.Q \mid R) \to (\mathsf{new}\ xy\colon U[v/w])(P \mid Q[v/z] \mid R) \qquad \text{(R-Com)}$$
$$(\mathsf{assume}\ \varphi)(\mathsf{assert}\ \varphi.P \mid Q) \to P \mid Q \qquad \text{(R-Assert)}$$
$$\frac{P \to Q}{(\mathsf{assume}\ \varphi)P \to (\mathsf{assume}\ \varphi)Q} \qquad \frac{P \to Q}{(\mathsf{new}\ xy\colon T)P \to (\mathsf{new}\ xy\colon T)Q} \qquad \text{(R-Assume, R-Res)}$$
$$\frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \Rrightarrow P' \qquad P' \to Q' \qquad Q' \Rrightarrow Q}{P \to Q} \qquad \text{(R-Par, R-Heat)}$$

Figure 2: Operational semantics

On what concerns heating, the rules in the first two lines are standard in the pi calculus, those in the following three lines manipulate assume and assert processes, as well as linear logic formulae, in the expected way. The last line introduces the only truly directional rule, allowing the scope of an assumption to encompass another process. The reason why the rule is not bidirectional is because we want to keep assertions in the scope of assumptions; take for example a process $P$ of the form $(\mathsf{assume}\ A)(\mathbf{0} \mid \mathsf{assert}\ A.\mathbf{0})$. We have that $P$ reduces in one step to $\mathbf{0}$, but $(\mathsf{assume}\ A)\mathbf{0} \mid \mathsf{assert}\ A.\mathbf{0}$ is stuck. With assume, and unlike scope extrusion, i.e., $(\mathsf{new}\ xy)P \mid Q \equiv (\mathsf{new}\ xy)(P \mid Q)$, we do not have bound variables to control the application of the rule. The last rule in the figure allows to expand a recursive type, paving the way applications of rule R-Com. Notice that we do not mention the usual sideconditions, e.g., that $x,y \notin \mathrm{fv}(Q)$ in the scope extrusion rule, since the variable convention can be assumed to provide this guarantee.

In the example, by heating, the $(\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))$ can be extended to encompass the **Store** process, and then moved to a position before session creation $(\mathsf{new}\ s_1s_2)$ to allow the interaction between the Client and the Store on channel $s_1s_2$, via the R-Com rule.

$$(\mathsf{new}\ s_1s_2)(\mathbf{Client_1} \mid \mathbf{Store}) \Rrightarrow$$
$$(\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))(\mathsf{new}\ s_1s_2)(s_1!\mathrm{p}.s_1!\mathrm{c}.s_1!100.\mathbf{0} \mid s_2?p.s_2?c.s_2?a.\mathbf{Charge}(c,a)) \to\to\to$$
$$(\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))\mathbf{Charge}(\mathrm{c},100)$$

Next, the process is ready to perform the communication between $\mathbf{Bank_1}$ and **Store**. Rule R-Assert matches the assume with the assert, and the process is concluded.

$$(\mathsf{new}\ r_1r_2)(\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))(\mathbf{Charge}(\mathrm{c},100) \mid \mathbf{Bank_1}) \equiv$$
$$(\mathsf{assume}\ \mathit{charge}(\mathrm{c},100))(\mathsf{new}\ r_1r_2)(\mathbf{Charge}(\mathrm{c},100) \mid r_1?y.y?c.y?a.\mathsf{assert}\ \mathit{charge}(c,a) \mid \mathbf{Bank_1}) \to\to\to$$
$$\mathsf{assume}\ \mathit{charge}(\mathrm{c},100)\ \mathsf{assert}\ \mathit{charge}(\mathrm{c},100).\mathbf{0} \mid (\mathsf{new}\ r_1r_2)\mathbf{Bank}_1 \to (\mathsf{new}\ r_1r_2)\mathbf{Bank_1}$$

Clearly, if $\mathbf{Store_1}$ is used, the reduction will yield a process where the assumption and the assertion do not match.

$$(\mathsf{new}\, r_1 r_2)(\mathsf{new}\, s_1 s_2)(\mathbf{Client} \mid \mathbf{Store_1} \mid \mathbf{Bank_1}) \to \cdots \to \\ \mathsf{assume}\, \mathit{charge}(c,100)\, \mathsf{assert}\, \mathit{charge}(c,110).\mathbf{0} \mid (\mathsf{new}\, r_1 r_2)\mathbf{Bank_1} \not\to$$

In turn, if $\mathbf{Store_2}$ replaces $\mathbf{Store_1}$ in the above process, then we reach a situation where one assertion is left unmatched.

$$(\mathsf{new}\, r_1 r_2)(\mathsf{new}\, s_1 s_2)(\mathbf{Client} \mid \mathbf{Store_2} \mid \mathbf{Bank_1}) \to \cdots \to \mathsf{assert}\, \mathit{charge}(c,100).\mathbf{0} \mid (\mathsf{new}\, r_1 r_2)\mathbf{Bank_1} \not\to$$

These two processes are stuck due to assume/assert problems — in both cases we find an assert for which no corresponding assume exists in the enclosing scope — and will be identified as unsafe by the typing system.

If somehow the client wants to be charged twice, then it can $\mathsf{assume}\, \mathit{charge}(\mathrm{c},100)$, twice in a row. Alternatively it may utilise a more compact variant by using joining (tensor).

$$\mathbf{Client_2} = (\mathsf{assume}\, \mathit{charge}(\mathrm{c},100) \otimes \mathit{charge}(\mathrm{c},100))s_1!\mathrm{p}.s_1!\mathrm{c}.s_1!100.\mathbf{0}$$

Then, by taking advantage of the heating rule that allows breaking the $(\otimes)$, as well as reduction underneath assumptions, we can easily see that:

$$(\mathsf{new}\, r_1 r_2)(\mathsf{new}\, s_1 s_2)(\mathbf{Client_2} \mid \mathbf{Store_2} \mid \mathbf{Bank_1}) \to \cdots \to (\mathsf{new}\, r_1 r_2)\mathbf{Bank_1}$$

We conclude this section by defining what we mean by a safe process. First we introduce the notion of *canonical processes*. A process is in canonical form if it is of the form:

$$(\mathsf{new}\, x_1 y_1 : T_1) \cdots (\mathsf{new}\, x_k y_k : T_k)(\mathsf{assume}\, A_1) \cdots (\mathsf{assume}\, A_m)(P_1 \mid \cdots \mid P_n) \qquad \text{with } k,m \geq 0, n > 0$$

and every $P_i$ is *neither* a new, nor an assume nor a parallel composition. A simple induction on the structure of processes easily allows us to conclude that all processes can be heated to a process in canonical form.

Then, we say that a process $Q$ is *safe* if, for all processes $P$ in the canonical form above such that $Q \Rrightarrow P$ and every $P_i$ of the form $\mathsf{assert}\, B_i.R_i$, there is a $1 \leq j \leq m$ such that $B_i = A_j$. In other words, safe processes do not get stuck at assertion points. The next section introduces a type assignment system that guarantees that processes typable under unrestricted contexts are safe.

Notice that each $A_i$ and $B_i$ are atomic formulae; if not, then the heating relation may "break" the tensors $(\otimes)$ and eliminate the identities $(\mathbf{1})$, so that in the end we may match assumptions on atomic formulae against assertions on atomic formulae.

## 3 Typing system

The syntax of types is presented in Figure 3. Let product, ccard and nat be the types of the products sold by the store, credit cards, and natural numbers respectively (all denoted by unit in the figure). The types of the two ends $r_1 r_2$ of the Client-Store channel, and also of the Bank-Store channel $s_1 s_2$, are as follows.

$s_1$ : lin!product.lin!ccard.lin!nat.end  $s_2$ : lin?product.lin?ccard.lin?nat.end

$r_1$ : $\mu\alpha$.un?(lin?ccard.lin?nat.end).$\alpha$  $r_2$ : $\mu\alpha$.un!(lin?ccard.lin?nat.end).$\alpha$

| | | | | | |
|---|---|---|---|---|---|
| $q$ ::= | | *Qualifiers:* | | $q\,p$ | qualified session |
| | $\mathsf{lin}$ | linear | | $\{x\colon T \mid \varphi\}$ | refinement |
| | $\mathsf{un}$ | unrestricted | | $\alpha$ | type variable |
| $p$ ::= | | *Session types:* | | $\mu\alpha.T$ | recursive type |
| | $?x\colon T.T$ | receive | $\Gamma$ ::= | | *Contexts:* |
| | $!x\colon T.T$ | send | | $\cdot$ | empty |
| $T$ ::= | | *Types:* | | $\Gamma, x\colon T$ | type assumption |
| | $\mathsf{unit}$ | unit | | $\Gamma, \varphi$ | formula |
| | $\mathsf{end}$ | termination | | | |

Figure 3: The syntax of types and typing contexts

In types, as in processes, ! means output and ? means input, $\mathsf{end}$ denotes a channel on which no further interaction is possible, and the $\mu$ construct is used to write recursive types. Qualifiers $\mathsf{lin}$ and $\mathsf{un}$ are used to control the number of threads holding references to the channel end: exactly one in the $\mathsf{lin}$ case, zero or more for the $\mathsf{un}$ case. The Client-Store channel is $\mathsf{lin}$ at all times, so that a third process cannot interfere in the interaction. The Bank-Store channel is $\mathsf{un}$ at all times, so that multiple stores may connect to the bank. Such a $\mathsf{un}$ channel is used to pass a $\mathsf{lin}$ channel (of type $\mathsf{lin?ccard.lin?nat.end}$), thus establishing a private channel between the Bank and the Store. In our example, we assume that the private Client-Store channel was created via a similar mechanism, based on some shared channel provided by the store. It should be easy to see that the type $\mathsf{lin!product.lin!ccard.lin!nat.end}$ of the $s_1$ end of the channel naturally describes the **Client**'s interaction $s_1!\mathsf{p}.s_1!\mathsf{c}.s_1!100.\mathbf{0}$, and that the type $\mathsf{un!(lin?ccard.lin?nat.end)}\ldots$ closely explains the **Store**'s interaction $r_1?y.y?c.y?a\ldots$

The above typing context is correct for process **Client** | **Store** | **Bank**, but it remains so even if one replaces **Store** by $\mathbf{Store_1}$ or by $\mathbf{Store_2}$, since in both of these cases the usage of the channels match the prescribed behavior. Thus, traditional session types are not enough to control and discipline the use of resources.

In order to incorporate logical information into session types, the syntax is augmented with logical refinements, $\{x\colon T \mid \varphi\}$. Further, and in order for formulae $\varphi$ to be able to refer to data appearing "previously" in types, we name the object of communication: in type $q?x\colon T.U$ we allow type $U$ to refer to the value received before via variable $x$. Types can be refined with the exact same formulae used for asserting and assuming in processes. For example, the types for channels $s_1$ and $r_1$ can be logically refined in such a a way that the amount $x$ to be charged is subject to "permission" $\mathit{charge}(c,x)$, where $c$ denotes the credit card number received in a previous communication.

$$s_1\colon \mathsf{lin!}p\colon \mathsf{product.lin!}c\colon \mathsf{ccard.lin!}a\colon \{x\colon \mathsf{nat} \mid \mathit{charge}(c,x)\}.\mathsf{end}$$
$$r_1\colon \mu\alpha.\mathsf{un?}y\colon (\mathsf{lin?}c\colon \mathsf{ccard.lin?}a\colon \{x\colon \mathsf{nat} \mid \mathit{charge}(c,x)\}.\mathsf{end}).\alpha$$

We will get back to our running example after introducing the type system.

For *recursive types*, type variable $\alpha$ occurs bound in type $\mu\alpha.T$. Such types are required to be *contractive*, i.e., containing no subexpression of the form $\mu\alpha_1\ldots\mu\alpha_n.\alpha_1$. We further require types not to contain subexpressions of the form $\mu\alpha_1\ldots\mu\alpha_n.\{x\colon T \mid \varphi\}$, so that the only interesting recursive types

*The dual of a type,* $\overline{T} = T$

$$\overline{q?x\colon T.U} = q!x\colon T.\overline{U} \qquad \overline{q!x\colon T.U} = q?x\colon T.\overline{U} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{\mu a.T} = \mu a.\overline{T} \qquad \overline{a} = a$$

*Unrestricted types and contexts,* $\mathsf{un}(T)$ *and* $\mathsf{un}(\Gamma)$

$$\mathsf{un}(\mathsf{unit}) \qquad \mathsf{un}(\mathsf{end}) \qquad \mathsf{un}(\mathsf{un}\, p)$$
$$\mathsf{un}(\cdot) \qquad \mathsf{un}(\Gamma, x\colon T) \text{ if } \mathsf{un}(\Gamma) \text{ and } \mathsf{un}(T)$$

*Well-formed formulae,* $\Gamma \vdash_{\mathsf{wf}} \varphi$*, well-formed types,* $\Gamma \vdash_{\mathsf{wf}} T$*, and well-formed contexts,* $\vdash_{\mathsf{wf}} \Gamma$

$$\frac{\mathrm{fv}(\varphi) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{wf}} \varphi} \qquad \frac{\mathrm{fv}(T) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{wf}} T} \qquad \vdash_{\mathsf{wf}} \cdot \qquad \frac{\vdash_{\mathsf{wf}} \Gamma \qquad \Gamma \vdash_{\mathsf{wf}} T}{\vdash_{\mathsf{wf}} \Gamma, x : T} \qquad \frac{\vdash_{\mathsf{wf}} \Gamma \qquad \Gamma \vdash_{\mathsf{wf}} \varphi}{\vdash_{\mathsf{wf}} \Gamma, \varphi}$$

Figure 4: Type duality, unrestricted predicates, and well formed predicates

are session types. We leave the treatment of recursive refinement types for future work, where they may represent the introduction of persistent formulae, i.e., the exponentials of linear logic. We again follow Barendregt's variable convention, this time on type variables $\alpha$.

*Type equivalence* is a central ingredient in dependent type systems. Here we stick to a rather simple notion. The equivalence relation of formulae is the smallest equivalence relation, denoted by $\equiv$, containing the axioms $\varphi_1 \otimes \varphi_2 \equiv \varphi_2 \otimes \varphi_1$ and $\varphi \otimes \mathbf{1} \equiv \varphi$. For types, we include in the equivalence relation a recursive type $\mu\alpha.T$ and its unfolding $T[\mu\alpha.T/\alpha]$, as well as refinement types that differ on equivalent formulae only. The definition, omitted, is co-inductive.

*Duality* plays a central role in the theory of session types. The two ends of a channel are supposed to be of a dual nature at certain points in typing derivations, namely at scope restriction $(\mathsf{new}\, xy\colon T)P$. Examples include the types for variables $s_1$ and $s_2$, as well as those for variables $r_1$ and $r_2$ above. The definition is in Figure 4. Duality is defined only for session types (input, output, end, and recursion); in particular it is undefined for refinement types in very much the way as it is undefined for unit [18].

*Typing contexts* are defined in Figure 3 and include type assumptions for variables, $x\colon T$, as well as formulae $\varphi$ known to hold. The domain of a context $\Gamma$, denoted $\mathrm{dom}(\Gamma)$, is defined as $\{x \mid x\colon T \in \Gamma\}$.

Types (and contexts) can be classified as *unrestricted* or *linear*; we only need the first notion; the definition is in Figure 4. Unrestricted types, denoted $\mathsf{un}(T)$, are unit, end and $\mathsf{un}\, p$ for all $p$. Unrestricted contexts may contain unrestricted types only, in particular they cannot contain formulae (for these are linear).

Formulae may contain program variables. Because types may include formulae, types may contain free program variables. Formulae and types are well formed with respect to a context if their free variables are in the domain of the context. Contexts contain formulae and types. Formulae and types appearing in a context must be well formed with respect to the "initial" part of the context. The definitions of well formed contexts is in Figure 4. In particular, our system does not include (implicitly or explicitly) the exchange rule; context $x\colon \mathsf{unit}, A(x)$ is well formed but $A(x), x\colon \mathsf{unit}$ not.

Central to our type system is the *context split* operator that distributes incoming formulae and linear types to one of the output contexts while duplicating incoming unrestricted types to both the output contexts. The definition, a straightforward extension of the one in [18] that can now handle formulae, is in Figure 5. Formulae in contexts are handled very much like linear type assumptions: there is one rule to "send" the formula (or type assumption) to the left context and one rule to send it to the right.

*Context split,* $\Gamma = \Gamma \circ \Gamma$

$$\emptyset = \emptyset \circ \emptyset \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash_{\mathsf{wf}} \mathsf{lin}\, p}{\Gamma, x\colon \mathsf{lin}\, p = (\Gamma_1, x\colon \mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_2 \vdash_{\mathsf{wf}} \mathsf{lin}\, p}{\Gamma, x\colon \mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x\colon \mathsf{lin}\, p)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \mathsf{un}(T)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \circ (\Gamma_2, x\colon T)} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_1 \vdash_{\mathsf{wf}} \varphi}{\Gamma, \varphi = (\Gamma_1, \varphi) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Gamma_2 \vdash_{\mathsf{wf}} \varphi}{\Gamma, \varphi = \Gamma_1 \circ (\Gamma_2, \varphi)}$$

*Context update,* $\Gamma + x\colon T = \Gamma$

$$\frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash_{\mathsf{wf}} T}{\Gamma + x\colon T = \Gamma, x\colon T} \qquad \frac{\mathsf{un}(T)}{(\Gamma, x\colon T) + x\colon T = (\Gamma, x\colon T)}$$

Figure 5: Context split and context update

There are however new assumptions, $\Gamma \vdash_{\mathsf{wf}} \varphi$ and $\Gamma \vdash_{\mathsf{wf}} \mathsf{lin}\, p$, meant to guarantee that the output of context splitting are well-formed contexts. The context update operator is used to update the type of a channel, after its prefix has been used. It is used in the typing rules for input and output processes.

We are finally in a position to introduce the type system; the rules are in Figure 6. Sequents for extracting formulae from contexts are of the form $\Gamma \vdash \varphi$; sequents for values are of the form $\Gamma \vdash v\colon T$, and for processes of the form $\Gamma \vdash P$. The rules for formulae should be easy to understand. All our rules make sure that at the leaves of derivations there are only well-formed, unrestricted contexts, so as to make sure all linear entities (formulae and types) are used in a derivation. The rules for values follow a similar pattern; they include conventional rules for refinement introduction and for type conversion. The first six rules for processes are taken from [18]. For instance, the rule for output splits the incoming context in three parts, one to type the subject $x$ of communication, the other to type the object $v$, and the third to type the continuation process $P$. The context for $P$ is updated with the new type for $x$, that is the continuation type $U$ with the appropriated substitution applied.

For example, in order to type the final part $s_1!100.\mathbf{0}$ of the **Client$_1$** process under context:

$$c\colon \mathsf{ccard}, s_1\colon \mathsf{lin}!a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}.\mathsf{end}, \mathit{charge}(c,100),$$

we split the context in three parts: $c\colon \mathsf{ccard}, s_1\colon \mathsf{lin}!a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}.\mathsf{end}$ to type variable $s_1$, context $c\colon \mathsf{ccard}, \mathit{charge}(c,100)$ to type value 100 and context $c\colon \mathsf{ccard} + s_1\colon \mathsf{end}[100/a] = c\colon \mathsf{ccard}, s_1\colon \mathsf{end}$ to type the continuation process $\mathbf{0}$. From the context for value 100, we build the type $\{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}$ that matches the "initial" part of the type for $s_1$. Formula $\mathit{charge}(c,100)$ is introduced in the context via the typing rule for assume (see below).

The novelties of the type system are the rules for assume and assert, and should be easy to understand. Rule T-ASSUME adds to the context the formula assumed in the process. Rule T-ASSERT works in the opposite direction, removing from the context the assertion. Also novel to our type system are the three rule for the elimination of $\mathbf{1}$, $\otimes$ and refinement types. These rules work in the context, hence are rules for processes. The corresponding introduction rules work on the entities (types and formulae) extracted from the context, and are thus rules for formulae and values.

Back to the running example, let $B_2 = \mu\alpha.\mathsf{un}!y\colon (\mathsf{lin}?c\colon \mathsf{ccard}.\mathsf{lin}?a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c+10,x)\}.\mathsf{end}).\alpha$ be the type of a bank as seen from the side of the **Store$_1$** (the type of $r_2$). Even though we can derive

$$s_1\colon \mathsf{lin}!p\colon \mathsf{product}.\mathsf{lin}!c\colon \mathsf{ccard}.\mathsf{lin}!a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}.\mathsf{end},$$
$$s_2\colon \mathsf{lin}?p\colon \mathsf{product}.\mathsf{lin}?c\colon \mathsf{ccard}.\mathsf{lin}?a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c+10,x)\}.\mathsf{end}, r_2\colon B_2 \vdash \mathbf{Client} \mid \mathbf{Store_1}$$

*Typing rules for formulae,* $\Gamma \vdash \varphi$

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad \mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{1}} \qquad \frac{\vdash_{\mathsf{wf}} \Gamma_1, \varphi, \Gamma_2 \quad \mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, \varphi, \Gamma_2 \vdash \varphi} \qquad \frac{\Gamma_1 \vdash \varphi_1 \quad \Gamma_2 \vdash \varphi_2}{\Gamma_1 \circ \Gamma_2 \vdash \varphi_1 \otimes \varphi_2} \qquad (\text{T-}\mathbf{1}\text{I,T-FORM,T-}\otimes\text{I})$$

*Typing rules for values,* $\Gamma \vdash v : T$

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad \mathsf{un}(\Gamma)}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{\vdash_{\mathsf{wf}} \Gamma_1, x\colon T, \Gamma_2 \quad \mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x\colon T, \Gamma_2 \vdash x\colon T} \qquad \frac{\Gamma_1 \vdash \varphi[v/x] \quad \Gamma_2 \vdash v\colon T}{\Gamma_1 \circ \Gamma_2 \vdash v\colon \{x\colon T \mid \varphi\}} \qquad \frac{\Gamma \vdash v\colon T_1 \quad T_1 \equiv T_2}{\Gamma \vdash v\colon T_2}$$

(T-UNIT,T-VAR,T-REFI,T-CONV)

*Typing rules for processes,* $\Gamma \vdash P$

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad \mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2}{\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2} \qquad \frac{\Gamma \vdash_{\mathsf{wf}} T \quad \Gamma, x\colon T, y\colon \overline{T} \vdash P}{\Gamma \vdash (\mathsf{new}\, xy\colon T)P} \qquad \frac{\mathsf{un}(\Gamma) \quad \Gamma \vdash P}{\Gamma \vdash *P}$$

(T-END,T-PAR,T-RES,T-REP)

$$\frac{\Gamma_1 \vdash x\colon (q!y\colon T.U) \quad \Gamma_2 \vdash v\colon T \quad \Gamma_3 + x\colon U[v/y] \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!v.P} \qquad (\text{T-OUT})$$

$$\frac{\Gamma_1 \vdash x\colon (q?y\colon T.U) \quad (\Gamma_2, z\colon T) + x\colon U[z/y] \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash x?z.P} \qquad (\text{T-IN})$$

$$\frac{\Gamma_2 \vdash \varphi \quad \Gamma_1 \circ \Gamma_2 \vdash P}{\Gamma_1 \vdash (\mathsf{assume}\, \varphi)P} \qquad \frac{\Gamma_1 \vdash \varphi \quad \Gamma_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{assert}\, \varphi.P} \qquad (\text{T-ASSUME,T-ASSERT})$$

$$\frac{\Gamma \vdash P}{\Gamma, \mathbf{1} \vdash P} \qquad \frac{\Gamma_1, \varphi_1, \varphi_2, \Gamma_2 \vdash P}{\Gamma_1, \varphi_1 \otimes \varphi_2, \Gamma_2 \vdash P} \qquad \frac{\Gamma_1, x\colon T, \varphi[x/y], \Gamma_2 \vdash P}{\Gamma_1, x\colon \{y\colon T \mid \varphi\}, \Gamma_2 \vdash P} \qquad (\text{T-}\mathbf{1}\text{E,T-}\otimes\text{E,T-REFE})$$

Figure 6: Typing rules

we cannot derive $r_2\colon B_2 \vdash (\mathsf{new}\, s_1 s_2)(\mathbf{Client_1} \mid \mathbf{Store})$ for the types for $s_1$ and $s_2$ are not dual, because type $\{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}$ is not equivalent to $\{x\colon \mathsf{nat} | \mathit{charge}(c+10,x)\}$, as required by rule T-RES.

The case of $\mathbf{Store_2}$ is of a different nature, and in particular it is not typable due to the impossibility of a suitable context split. One would like to type $\mathbf{Store_2}$ under context:

$$s_2\colon \mathsf{lin}?p\colon \mathsf{product}.\mathsf{lin}?c\colon \mathsf{ccard}.\mathsf{lin}?a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}.\mathsf{end}, r_2\colon B_2'$$

where $B_2'$ is type $B_2$ above with $\mathit{charge}(c,x)$ replacing $\mathit{charge}(c+10,x)$. Typing the initial part of the process, using rule T-IN three times, we introduce in the context the following entries: $p\colon \mathsf{product}$, $c\colon \mathsf{ccard}$, and $a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x)\}$. Then, using refinement elimination rule, T-REFE, we convert the last entry in $a\colon \mathsf{nat}, \mathit{charge}(c,a)$. Now, in order to type the continuation $\mathbf{Charge}(c,x) \mid \mathbf{Charge}(c,x)$, we have to split the context, but there is one only formula $\mathit{charge}(c,a)$ in the incoming context, so that only one of the threads will be typable.

On the other hand, consider the case of $\mathbf{Client_2}$ above that assumes twice the capability $\mathit{charge}(\mathrm{c}, 100)$. By duality of sessions, the type of the value received by the store will also be refined with a double capability, $a\colon \{x\colon \mathsf{nat} | \mathit{charge}(c,x) \otimes \mathit{charge}(c,x)\}$. Then we use T-REFE followed by T-$\otimes$E to obtain $a\colon \mathsf{nat}, \mathit{charge}(c,a), \mathit{charge}(c,a)$, making possible the split $a\colon \mathsf{nat}, \mathit{charge}(c,a) \circ a\colon \mathsf{nat}, \mathit{charge}(c,a)$.

# 4 Main results

The central result of this paper follows from the lemmas for weakening, strengthening and substitution [18] extended to this system, as well as from basic properties of context splitting (details omitted).

**Lemma 1** (Weakening)**.** *If* $\Gamma \vdash P$ *and* $\mathsf{un}(T)$*, then* $\Gamma, x : T \vdash P$.

**Lemma 2** (Strengthening)**.** *If* $\Gamma, x : T \vdash P$*,* $\mathsf{un}(T)$ *and* $x \notin \mathrm{fv}(P)$*, then* $\Gamma \vdash P$.

**Lemma 3** (Substitution)**.** *If* $\Gamma_1 \vdash v : T$ *and* $\Gamma_2, x : T, \Gamma_3 \vdash P$*, then* $\Gamma_1 \circ (\Gamma_2, \Gamma_3[v/x]) \vdash P[v/x]$.

**Lemma 4** (Preservation for $\Rrightarrow$)**.** *If* $\Gamma \vdash P$ *and* $P \Rrightarrow Q$*, then* $\Gamma \vdash Q$.

**Theorem 5** (Preservation for $\to$)**.** *If* $\Gamma \vdash P$ *and* $P \to Q$*, then* $\Gamma \vdash Q$.

**Theorem 6** (Safety)**.** *If* $\Gamma \vdash P$ *and* $\mathsf{un}(\Gamma)$*, then P is safe.*

It should be easy to see that processes typable under arbitrary contexts may not be safe; take for example $A \vdash \mathsf{assert}\, A.\mathbf{0}$.

Finally, combining the two results above with a simple induction on the length of reduction we obtain the main result of the paper.

**Corollary 7** (Main Result)**.** *If* $\Gamma \vdash P$ *with* $\mathsf{un}(\Gamma)$ *and P reduces to Q in a finite number of steps, then Q is safe.*

The result states that processes typable under unrestricted contexts do not get stuck at assertion points (they may still block at input or output points, due to deadlock). Furthermore we also know that all assumptions are eventually matched; e.g., process $(\mathsf{assume}\, A)\mathbf{0}$ is not typable. In the case of typable processes it is therefore safe to erase all the assumptions and assertions from a process, so that there are no formulae at runtime.

For the cases in proofs involving formulae we make use of the notion of *canonical contexts*, that is, contexts containing no refinement types ($x\colon T \in \Gamma$ implies $T$ is not a refinement type) and whose formulae contain no connectives ($\varphi \in \Gamma$ implies $\varphi = A$). Contexts can be converted in a canonical form by using the cf function, defined on contexts, type assumptions, and formulae.

$$\mathrm{cf}(\cdot) = \cdot \qquad \mathrm{cf}(\Gamma, \varphi) = \mathrm{cf}(\Gamma), \mathrm{cf}(\varphi) \qquad \mathrm{cf}(\Gamma, x : T) = \mathrm{cf}(\Gamma), \mathrm{cf}(x : T)$$
$$\mathrm{cf}(x : \{y : T \mid \varphi\}) = \mathrm{cf}(x : T), \mathrm{cf}(\varphi[x/y]) \qquad \mathrm{cf}(x : T) = x : T \text{ if } T \text{ is not a refinement}$$
$$\mathrm{cf}(\mathbf{1}) = \cdot \qquad \mathrm{cf}(\varphi_1 \otimes \varphi_2) = \mathrm{cf}(\varphi_1), \mathrm{cf}(\varphi_2) \qquad \mathrm{cf}(A) = A$$

We then establish a result, $\Gamma \vdash P$ iff $\mathrm{cf}(\Gamma) \vdash P$, allowing to consider contexts in their canonical form.

# 5 Related Work, Conclusions and Future Plans

Refinements have been useful in verifying polymorphic contracts [1], security protocols [2], and with the improvements in satisfiability-modulo-theories (SMT) solvers for classical first-order logics with uninterpreted functions (such as [14]), can be integrated into type systems using off-the-shelf components as has been done for the language F# using the F7 typechecker [15].

In the context of sessions, Bonelli et al. system of correspondence assertions for process synchronization [4] is close to a basic form of refinement, as it allows labels to be used in the participant processes of a session to signify the starting and ending points of marked protocol sections. This type of protocol segmentation can be thought of as a basic assume/assert mechanism with conjunction, since multisets of labels can be used for the part equivalent to 'assert,' but still without the rich constructors and proof

system of a logic. Bocchi et al. introduce assertions in multiparty session types (session types allowing to describe interaction among multiple partners) [3]. Similarly to the system of Bonelli et al. [4], assertions are explicitly associated with session operations (in, out, branch, select). In contrast, our system introduces assertions as refinement types to be used at arbitrary places in a protocol; furthermore their system uses classical logic as opposed to linear logic.

The recent work by Toninho et al. [17] interprets session types within intuitionistic linear logic, obtaining (with some extensions) a dependent sessions type system for $\pi$-calculus. This system interprets session types as linear logic formulae, with input as $\multimap$ and output as $\otimes$, and stratifies the language into a $\pi$-calculus for communication and a functional language for proof objects, where the latter are opaque terms that (in our system) would correspond to proofs of refinements. However, their system does not consider *linear* refinements, i.e., linearity is restricted to the communication layer (the sessions). Although the aims of both systems are similar to an extent, we have taken a different approach, adopting session types without their linear-formulae interpretation, and focussing on the incorporation of fine-grained linear refinements which provide for a more delicate distinction between types. Moreover, we do not utilise proof-witnesses but rather implement proof search within the type system itself; then, using the heating relation, assumptions are manipulated at runtime in order to check assertions, which is essentially a procedure of cut-elimination.

The concept of names appearing in types was pioneered in the work by Yoshida on channel-dependent types for processes with code mobility [20], and was adapted to sessions in subsequent work [12]. In these systems there are no refinements, yet channel dependent types are shown to provide security guarantees by controlling which names may be used in communications between received code and host environment, which indicates that an integration with our system could provide even greater control over mobile code.

In summary, a theory of (linear) refinement types for sessions has not been hitherto proposed, marking the contribution of our system. As future work, it is interesting to consider sessions *as* linear refinements, and to extend our refinement language to a larger fragment of Linear Logic. We plan to investigate decidable type-checking, drawing inspiration from the techniques in [5, 16, 18], and by considering appropriate restrictions. Moreover, it would be interesting to examine the adaptations necessary for languages with (asynchronous) buffered semantics, where communications can be reordered, especially in the context of mobile session-typed processes [13], channel dependent types [12, 20], and multi-party sessions [10].

**Acknowledgements.** This work was supported by projects Interfaces, CMU-PT/NGN/0044/2008, and Assertion-types, PTDC/EIA-CCO/105359/2008.

## References

[1] João Belo, Michael Greenberg, Atsushi Igarashi & Benjamin Pierce (2011): *Polymorphic Contracts*. In: *Programming Languages and Systems, LNCS* 6602, Springer, pp. 18–37 doi:10.1007/978-3-642-19718-5_2.

[2] Karthikeyan Bhargavan, Cédric Fournet & Andrew D. Gordon (2010): *Modular verification of security protocol code by typing*. In: *POPL'10*, ACM, pp. 445–456, doi:10.1145/1706299.1706350. Available at 10.1145/1706299.1706350.

[3] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A theory of design-by-contract for distributed multiparty interactions*. In: *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, Springer, pp. 162–176 doi:10.1007/978-3-642-15375-4_12.

[4] Eduardo Bonelli, Adriana Compagnoni & Elsa Gunter (2005): *Correspondence Assertions for Process Synchronization in Concurrent Communications*. *Journal of Functional Programming* 15, pp. 219–247 doi:10.1017/S095679680400543X.

[5] Cormac Flanagan (2006): *Hybrid type checking*. In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'06, ACM, pp. 245–256 doi:10.1145/1111037.1111059.

[6] Tim Freeman & Frank Pfenning (1991): *Refinement types for ML*. In: *PLDI'91*, ACM, pp. 268–277 doi:10.1145/113446.113468.

[7] Jean-Yves Girard (1987): *Linear logic*. *Theoretical Computer Science* 50, pp. 1–102 doi:10.1016/0304-3975(87)90045-4.

[8] Andrew Gordon & Cédric Fournet (2009): *Principles and applications of refinement types*. TR 147, MSR doi:10.3233/978-1-60750-100-8-73.

[9] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In: *ESOP'98*, LNCS, Springer, pp. 122–138 doi:10.1007/BFb0053567.

[10] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL'08*, ACM, pp. 273–284, doi:10.1145/1328438.1328472. Available at doi:10.1145/1328438.1328472.

[11] Yitzhak Mandelbaum, David Walker & Robert Harper (2003): *An Effective Theory of Type Refinements*. In: *ICFP'03*, ACM, pp. 213–226 doi:10.1145/944705.944725.

[12] Dimitris Mostrous & Nobuko Yoshida (2007): *Two session typing systems for higher-order mobile processes*. In: *TLCA'07*, *LNCS* 4583, Springer, pp. 321–335 doi:10.1007/978-3-540-73228-0_23.

[13] Dimitris Mostrous & Nobuko Yoshida (2009): *Session-Based Communication Optimisation for Higher-Order Mobile Processes*. In: *TLCA'09*, *LNCS* 5608, Springer, pp. 203–218 doi:10.1007/978-3-642-02273-9_16.

[14] Leonardo de Moura & Nikolaj Bjorner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, *LNCS* 4963, Springer, pp. 337–340 doi:10.1007/978-3-540-78800-3_24.

[15] Microsoft Research: *F7: Refinement Types for F#*. `http://research.microsoft.com/en-us/projects/F7/`.

[16] Patrick M. Rondon, Ming Kawaguci & Ranjit Jhala (2008): *Liquid types*. In: *PLDI'08*, ACM, pp. 159–169 doi:10.1145/1375581.1375602.

[17] Bernardo Toninho, Luís Caires & Frank Pfenning (2011): *Dependent session types via intuitionistic linear type theory*. In: *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP'11, ACM, pp. 161–172 doi:10.1145/2003476.2003499.

[18] Vasco T. Vasconcelos (2012): *Fundamentals of Session Types*. *Information and Computation* 217, pp. 52–70, doi:10.1007/978-3-642-01918-0_4. Available at `http://www.di.fc.ul.pt/~vv/papers/vasconcelos_fundamental-sessions.pdf`. Earlier version in SFM'09, volume 5569 of LNCS, pages 158-186. Springer, 2009 doi:10.1007/978-3-642-01918-0_4.

[19] Philip Wadler & Robert Bruce Findler (2009): *Well-typed programs can't be blamed*. In: *ESOP'09*, Springer, pp. 1–16 doi:10.1007/978-3-642-00590-9_1.

[20] Nobuko Yoshida (2004): *Channel dependent types for higher-order mobile processes*. In: *POPL'04*, ACM, pp. 147–160 doi:10.1145/964001.964014.